アサーションを活用して不ぐあい箇所を即座に特定 後編

――複雑なシーケンスを簡潔に表現するテクニック

赤星博輝

デバイスの記事

前回は，SERE（Sequential Extended Regular Expression）と呼ばれる記法によって長いシーケンスを表現する方法を紹介した．今回は，シーケンスに名まえを付けて階層的に呼び出す方法や，シーケンスどうしの演算などについて解説する．また，アサーションのチェックを途中でキャンセルする方法についても説明する．（編集部）

最近は「検証が大事」と言われていますが，まだ，検証を設計の“おまけ”と考える人がいることも事実です．その一方で，設計対象が大規模化・複雑化している現状では，新しい検証手法などの導入が不可欠になっていることもまちがいありません．

それでは「検証」とは何なのでしょうか？ ここでは，「RTL記述が設計仕様どおり正しく設計されているかどうかをチェックすること」について考えてみます．このチェックを行うには，どのような方法があるのでしょうか？

RTL設計に対して，考えうるすべてのシミュレーション・パターンをチェックできればよいのですが，現実にはこれを行えないケースがほとんどです．その理由は，現在の大規模な設計では膨大な量のシミュレーション・パターンを作る必要があり，その作成がたいへんということと，その膨大なシミュレーション・パターンを使ってシミュレーションすると膨大な時間が必要になり，限られた設計期間ではシミュレーションを終了することができないということです．

ここでもう1歩進めて，検証は「シミュレーション・パターンを流すこと」ではなく，「設計時に潜り込んだバグを探し出すこと」が目的であると考えてみましょう．そうすると，シミュレーションだけでなく，設計レビューやアサーション，フォーマル・ベリフィケーション（形式的検証）などの手法も有効になってきます．バグ（虫）を捕まえるのに一つの方法だけを使うと，どうしても同じタイプのバグばかり発見する傾向があります．設計時に潜り込んださまざまなタイプのバグを見つけ出すには，違った視点からチェックすることが必要となります．残念ながら，「アサーションを覚えたらすべて安心」ということにはなりません．

それでは，アサーションにはどのような効果があるのでしょうか？ シミュレーションを漁船に例えると，シミュレーション・パターンはどこの漁場をどのように動くのかを決定することに対応します（**図1**）．これに対して，期待値チェックという網で魚を捕まえるのですが，出力を見るだけなので網の目が粗いと考えることができます．ここではすべての魚（バグ）を捕まえたいわけですから，網の目を細かくすることが有効になります．アサーションは，内部の信号やインターフェース部の動きをチェックします．つまり，より細かい網で魚（バグ）を捕まえることができます．

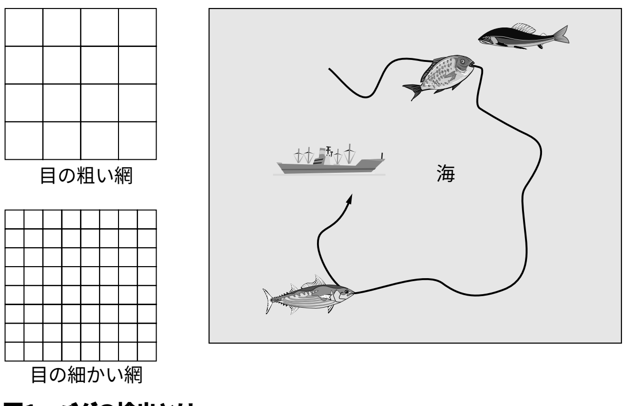


図1　バグの検出とは

漁船の行動に例えると，魚をとることがバグの検出と言える．シミュレーション・パターンは，漁船が海の上をどのように進むか（航路）を決定する．漁船は「期待値チェック」という網で魚をとるが，アサーションによって網の目を細かくすることができる．

アサーションなどを用いて網の目を細かくしても，シミュレーション・パターンの作りかたが悪いと，魚のいない場所を動き回って疲れるだけになりますし，網の目を細かくしすぎると抵抗が増えて進むのが遅く（シミュレーション時間が長く）なります．アサーションを使う場合には，よく考えて検証する必要があります．

● PSL/Sugarの三つの階層の概念を理解する

これまでは，PSL/Sugarを使った実例を挙げながら説明してきましたが，ある程度複雑なことをするためには基本となる考えかたを知っておく必要があります．今回はPSLの基本となる三つの階層（レイヤ）の説明から始めます．

PSLは正確には四つの階層を持っていますが，ここではアサーションを使用するために最低限必要な知識として，三つの階層についてのみ説明します．この階層の知識がないと，PSLの演算子とHDL（VHDL，Verilog HDL）の演算子のどちらを使うのかわからなくなったりするので，以下の三つの階層を頭に入れておきましょう．

- **論理式層（boolean layer）**
- **テンポラル層（temporal layer）**
- **検証層（verification layer）**

図2にこれらの階層と記述を示します．層とPSLの対応を見ていただければ，理解しやすいと思います．

1）論理式層

論理式層は，論理式を記述する層です．この層の記述が単体で存在することはなく，ほかの層から使用されます．論理式なので，ある時点でのプロパティを記述することになります．この層はHDLで記述されるものが基本となります．HDLとしてはVerilogフレーバとVHDLフレーバが使えるようになっています．

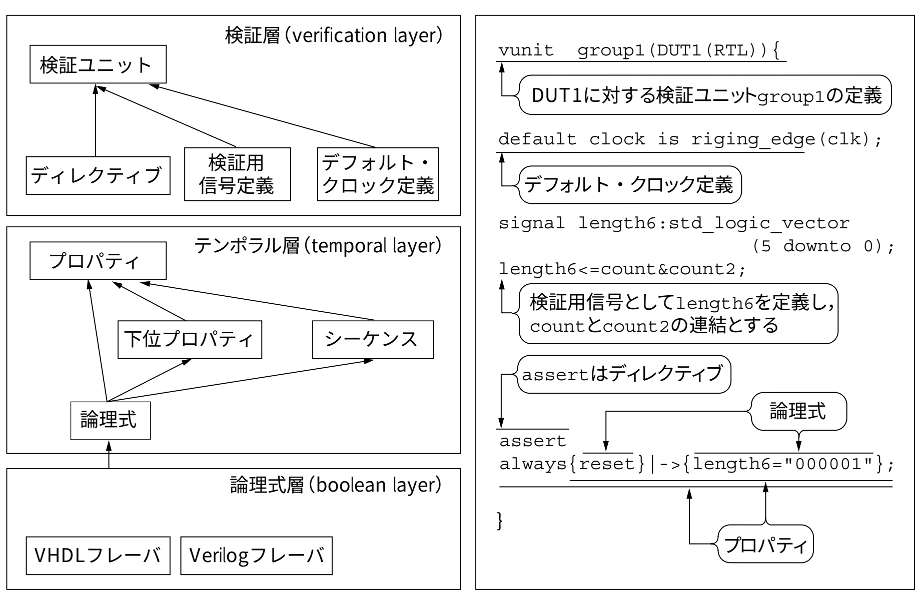
2）テンポラル層

テンポラル層では，複数クロックにまたがる回路のプロパティ（性質）を記述します．そのために，プロパティは以下の三つから構成されます．

- **論理式**
- **シーケンス**
- **下位のプロパティ**

論理式は論理式層で記述するものになり，先ほど述べたようにVerilogフレーバやVHDLフレーバなどを使用します．シーケンスは複数サイクルにまたがる回路の動作になり，前回説明したSERE（Sequential Extended Regular Expression）などで記述します．このシーケンス記述を使うと，シーケンスに名まえを付けて呼び出し，再利用したり，短いシーケンスを組み合わせて長いシーケンスを定義したり，シーケンスに対する演算を行うことで複雑なシーケンスを記述したりすることができます．また，プロパティも階層的に定義することができます．

これらの三つを組み合わせてプロパティの定義を行うことができます．注意する点として，論理式層は各HDLのフレーバを使用しますが，シーケンスなどではPSL/Sugarの



（a）階層　（b）記述

図2
PSL/Sugarの階層と実際の記述
アサーションを使用するために最低限必要な知識として，三つの階層の概念を理解する必要がある．論理式層は，論理式を記述する層である．テンポラル層では，複数クロックにまたがる回路のプロパティ（性質）を記述する．検証層では，検証ツール（シミュレータやプロパティ・チェッカなど）に「何をどのようにチェックしなさい」という指示を与えるためのディレクティブを記述したり，そのディレクティブをグループにまとめたり，チェックするときの基準となるデフォルト・クロックを定義したりする．

演算子を使うことが挙げられます．とくにVHDLは演算子がPSL/Sugarの演算子と異なるものが多いため，層の概念をきちんと理解しておく必要があります．

3）検証層

検証層では，検証ツール（シミュレータやプロパティ・チェッカなど）に「何をどのようにチェックしなさい」という指示を与えるためのディレクティブを記述したり，そのディレクティブをグループにまとめたり，チェックするときの基準となるデフォルト・クロックを定義したりします．このグループとしてまとめる単位を検証ユニット（verification unit）と呼びます．この検証ユニットでは検証用の信号の定義を行うことが可能で，新しい信号および論理式を記述することができます．**図2**の`length6`の信号のように，二つの信号を連結することでプロパティを容易に記述することもできます．

● シーケンスに名まえを付けたり，演算したりできる

前回はSEREについて説明しました．SEREはシーケンスを記述するためのものでした．ここでシーケンスとは，複数クロックにまたがってチェックする条件と考えてください．例えば，リクエスト信号が'1'になった次のサイクルでアクノリッジ信号が'1'になるといった条件です．

PSL/Sugarでシーケンスを記述する方法の全体像は以下のとおりです．

1）前回説明したSERE（{}で囲まれる記述）

● クロックの区切り（;）

例えば，「信号aが真になって，次のサイクルで信号bが真になる」というシーケンスは，`{a ; b}`と書くことができます．

● 連続繰り返し（[*]）

例えば，「信号aが2サイクル連続で真になる」というシーケンスは，`{a [*2]}`と書くことができます．

● 非連続繰り返し（[=]，[->]）

例えば，「信号aが2サイクル真になる（連続でなくてもよい場合）」というのは，`{a [=2]}`または`{a [->]}`と書くことができます（この二つの記述の違いについては，本誌2004年12月号，pp.130-135の本記事の中編を参照）．

2）シーケンスの名まえ付け

シーケンスに名まえを付けて，階層的にシーケンス（プロパティ）定義を行えます．これにより，再利用性や保守性が高まります．

3）シーケンスに対する演算

複数のシーケンスに対して演算を定義することができます．シーケンスの名まえ付けと組み合わせることで，再利用性や保守性をさらに高めることができます．

今回は，シーケンスの名まえ付けとシーケンスに対する演算について説明します．

● シーケンスを組み合わせて複雑な仕様を簡潔に記述

PSL/Sugarではシーケンスを定義し，呼び出すことができる枠組みがあります．

まず，定義は以下の形になります．

```
sequence シーケンスの名まえ is シーケンス定義 ;
```

それでは実際にシーケンスを書いてみます．3ビットの信号`count`が1，2となるシーケンスを`SA1_2`，信号`count`が3，4となるシーケンスを`SA3_4`として定義すると，

```
sequence SA1_2 is {count = "001" ; count
                                = "010" };
sequence SA3_4 is {count = "011" ; count
                                 = "100" }
```

になります．これらの動作を**図3（a），（b）**に示します．こ

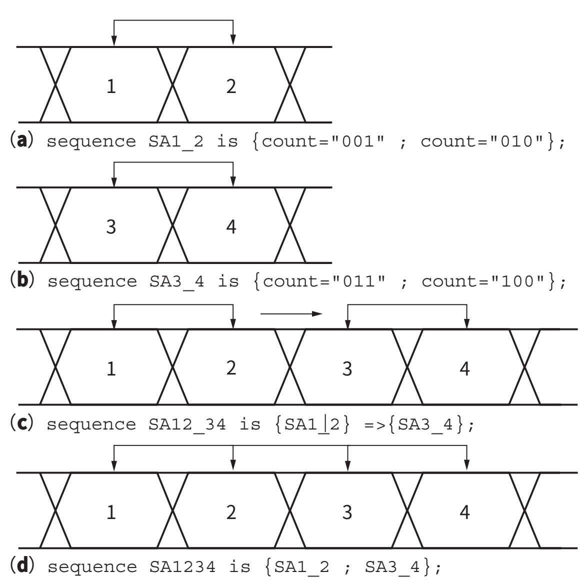


図3　シーケンス定義とその利用

シーケンスを定義することで，アサーションの定義や複雑なシーケンスをコンパクトに記述できる．再利用しやすくなり，また，アサーションが変更になった場合も記述の部分的な修正ですませることができる．例えばチェックする値が変わった場合などは，シーケンス定義を変更するだけで対応できることが多い．

れにより，アサーション定義でこのシーケンスを呼び出すことができます．

例えば，「3ビットの信号countが最初のサイクルで1に，次のサイクルで2になったならば，その次のサイクルでcountは3に，さらにその次のサイクルでcountは4になる」というプロパティをアサーションとして登録するには，以下のように記述します（動作は**図3（c）**に示す）．

```
sequence SA12_34 is {SA1_2} |=>
                            {SA3_4} ;
```

名まえ付きのシーケンスはSEREを記述するときに使用できます．例えば，SA1_2とSA3_4を組み合わせて以下のSA1234のようなより長いシーケンスを定義することが可能となります（動作は**図3（d）**に示す）．

```
sequence SA1234  is  {SA1_2 ; SA3_4};
```

この名まえ付きシーケンスによって，複数のシーケンスを組み合わせたプロパティ定義が可能となり，重複したシーケンス記述をまとめることによる記述量の削減や再利用性の向上を実現できます．また，記述量を削減できるということは，それだけ人為的なミスが入り込む余地が少なくなるということでもあります．積極的に使っていきましょう．

シーケンスの名まえをきちんと考えると，シーケンスを見ただけで何をチェックする記述かわかりやすくなります．例えばABCという名まえだと，見ただけでは何をチェックする記述なのかわかりません．一方，wait_ackなどという名まえを付ければ，理解しやすくなります．可読性を考慮して命名しておけば，そのあとの再利用にもつながります．

記述のポイントは，あらかじめ検証項目を考えて，これらをきちんと整理しておくことです．検証項目を整理することで，シーケンスを再利用することが容易になり，検証項目の抜けや漏れも少なくなります．

● シーケンスにパラメータを付ける

先ほど説明した名まえ付きシーケンスでは，count信号に対するシーケンスとして定義しました．ただし，この定義では信号名を変更できないため，重複したシーケンス記述をまとめる効果は期待できますが，再利用性という意味では限界があります（異なったモジュールを設計するときに，前に設計したモジュールと同じ信号名で同じプロパティをチェックすることがどのくらいあるかを考えていただきたい）．そこで，PSL/Sugarではシーケンスにパラメータを与えられるようになっています．記述は以下の形になります．

```
sequence シーケンスの名まえ（パラメータ・リスト）
                         is シーケンス定義 ;
```

パラメータとしては，定数，ブール式，シーケンスを与えることができます．ビット幅を持つ信号をパラメータにできないことに注意が必要です．**リスト1**にブール式とシーケンスの記述例を示します．

● シーケンスに対して演算を施す

名まえ付きシーケンスを使うことによって記述量を削減できますが，実はまだまだ便利な使いかたがあります．

図4のようにseqAとseqBというシーケンスを作っておけば，{seqA ; seqB}のように連結することができます．では，同時にseqAとseqBをチェックしたい場合にはどうしたらよいのでしょうか？ これまでの知識では再利用できないので，**図4**の左下（seqMixA）のような新しい記述を作成する必要があります．しかし，これから説明するシーケンスに対する演算を利用すると，**図4**の右下（seqMixB）のように前に作ったシーケンスを利用して簡単に記述することが可能となります．

ここでは，シーケンスに対する五つの演算を覚えましょう．連結も加えたシーケンスに対する演算を**表1**にまとめ

リスト1 シーケンスにパラメータを与える場合の記述例

```
sequence S1(boolean b1,b2) is { {b1} ; {b2} };
property A1 is always {count = "01"} |=> S1(count = "10", count = "11");
```

（**a**）ブール式の記述例

```
sequcnce S2 is {count = “10”};                    // シーケンス定義
sequence S3(sequence sa) is {sa};                  // シーケンスを使ったシーケンス定義
property A2 is always {count = “01”} |=> {S3(S2)};  // パラメータ付きシーケンスの呼び出し
```

（**b**）シーケンスの記述例

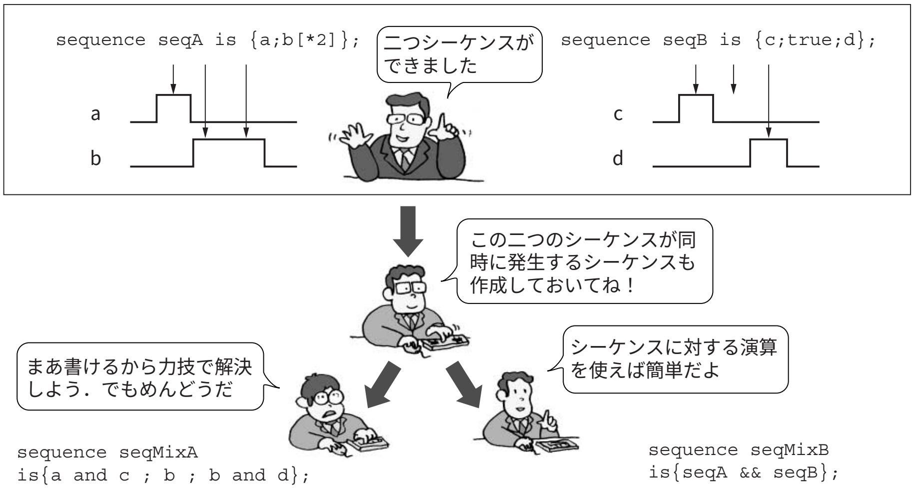


図4
シーケンスの演算が行えると便利
あるまとまりで再利用できないと，プロパティの記述作業が楽にならない．PSLではシーケンスの演算などを用意して，再利用しやすくしている．

ました．

1）シーケンスの連結（;）

最初に紹介するのは，すでに説明しているシーケンスの連結です．複数のシーケンスを連結するときに使用します．**図5（a）**の2サイクルのシーケンスS1と3サイクルのシーケンスS2を連結した `S1 ; S2` は，S1の最後のサイクルの次のサイクルでS2が実行されることになります．利用法としては，個々のテストで使用したシーケンスを統合して再利用したり，長いシーケンスを意味のあるまとまりごとに定義するといったことが可能となります．

2）シーケンスの融合（:）

2番目に紹介するのはシーケンスの融合です．先ほどのシーケンスの連結との違いは，1サイクル，オーバラップすることです．このオーバラップすることが融合と呼ばれる理由です．**図5（b）**の例を見ると，S1の最後のサイクルとS2の最初のサイクルが重なって融合されています．単純に連続する場合との違いは，このオーバラップのある，なしになります．

表1　シーケンスの演算

演　算	記　述	S1とS2のシーケンス長	意　味
連結	`S1 ; S2`	違ってもOK	S1が終わった次のサイクルでS2がスタートするシーケンス
融合	`S1 : S2`	違ってもOK	S1の最後のサイクルでS2がスタートするシーケンス
OR	`S1 | S2`	違ってもOK	S1またはS2のどちらかが成立するシーケンス
長さ不一致のAND	`S1 & S2`	違ってもOK	S1とS2の両方が成立するシーケンス
長さ一致のAND	`S1 && S2`	一致のみ	長さの同じS1とS2の両方が成立するシーケンス

3）シーケンスのOR（|）

3番目の演算はシーケンスのORです．二つのシーケンスのどちらかが成り立つという記述を行うときに利用できます．**図6**で説明すると，`{S1} | {S2} ; a` というシーケンスが成立するのは，三つの場合があります．すなわち，「S1が成り立った場合」，「S2が成り立った場合」，「S1とS2が成り立った場合」です．S1のみまたはS2のみが成り立った場合には，次のサイクルで信号aが真というシーケンスを記述していることになります．**図6**のタイミング・チャートのいちばん右端のS1とS2が成り立った場合には，最初に成り立ったシーケンス（この場合はS1）で評価を行います．

4）シーケンスの長さ不一致のAND（&）

4番目はシーケンスのANDです．シーケンスの長さが違っても利用できます．シーケンスの長さとは，チェックするサイクル数です．例えば，**図7（a）**のシーケンスS1は2サ

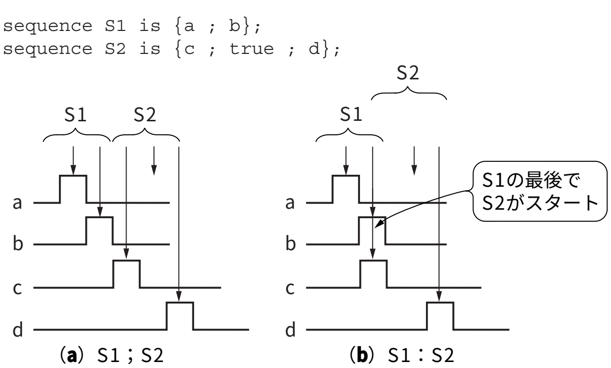


図5　シーケンスの連結と融合
連結は，前のシーケンスが終わった次のサイクルから始まる．これに対してシーケンスの融合は，前のシーケンスの最終サイクルと，次のシーケンスの最初のサイクルが重なる．この二つを使い分けることで，さまざまなシーケンスを細かい単位で管理できる．

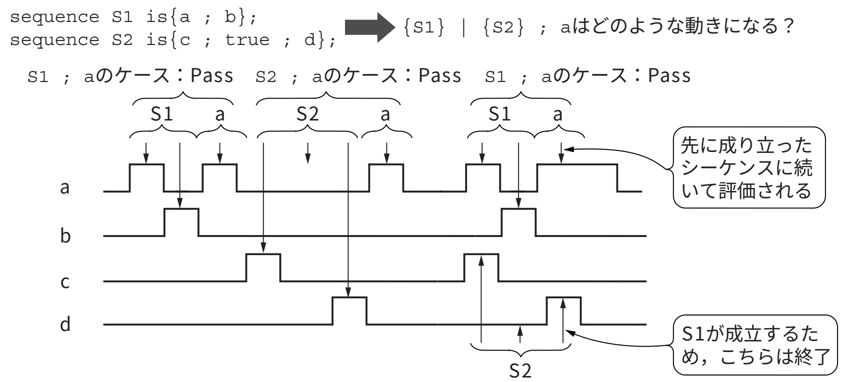


図6
シーケンスのOR
シーケンスのORは，二つのシーケンスのどちらかが成り立つことを記述する．シーケンスの長さが違うときに両方のシーケンスが成り立った場合は，早く成り立ったほうのみが有効になることに注意．

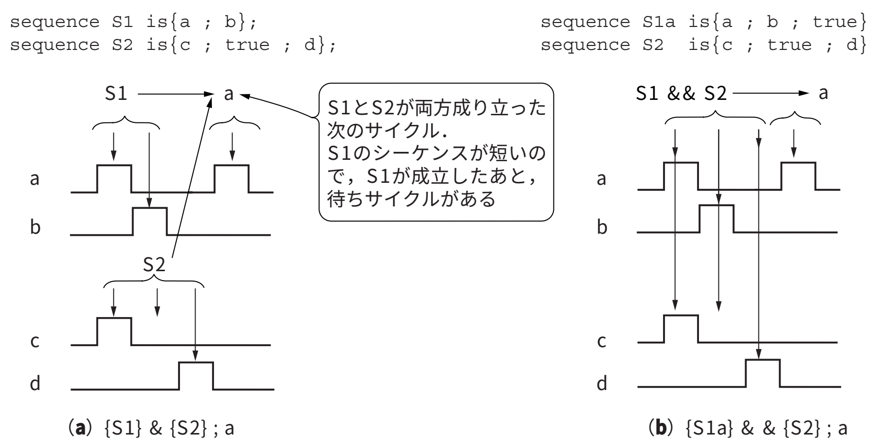


図7
シーケンスのAND
シーケンスの長さが同じ場合には&&を使う．シーケンスの長さが異なる場合は&を使う．シーケンスの長さが異なる場合，長いシーケンスに合わせて次の判断を行うので，短いほうから見ると待ちサイクルがあるように見える．

イクル，シーケンスS2は3サイクルになります．このときの二つのシーケンスが成り立つことを記述するときに，シーケンスのAND（&）を使用できます．**図7（a）**の`{S1} & {S2} ; a`の例では，「S1とS2の両方のシーケンスが成立した」というのがS2のシーケンスの成立時になるので，そのS2の成り立った次のサイクルで「信号aが真」というシーケンスを記述しています．

5）シーケンスの長さ一致のAND（&&）

5番目もシーケンスのANDですが，こちらはシーケンスの長さが一致している必要があります）．二つのシーケンスの長さが同じ場合，&&という記述を利用します（**図7（b）**）．長さが同じものはこちらを使うべきでしょう．

● シーケンスの完了状況の異なる二つの演算子がある

PSL/Sugarでは，強い演算子（strong operator）と弱い演算子（weak operator）があります．今回は，`|=>` や `|->` を例に説明してみます．これまでは，

```
シーケンスA  |->  シーケンスB
```

で「シーケンスAならばシーケンスBである（このとき，オーバラップあり）」という意味になり，

```
シーケンスA  |=>  シーケンスB
```

で，「シーケンスAならばシーケンスBである（オーバラップなし）」という意味になります．

実はこれらは弱い演算子と呼ばれています．理由は，右側のシーケンスBはかならず終わる必要がないためです．これに対して，強い演算子は，次のように最後に ! を付けます．

```
シーケンスA  |->  シーケンスB !
シーケンスA  |=>  シーケンスB !
```

この場合，右側のシーケンスBがかならず完了する必要があります．

右側のシーケンスが完了しないケースを考えてみましょう．例えば，銀行の窓口のようなものを考えます（**図8**）．このとき，お客さんが少なくて窓口がすいていれば，処理（預け入れや引き出しなど）に対する要求（req）をすぐに実

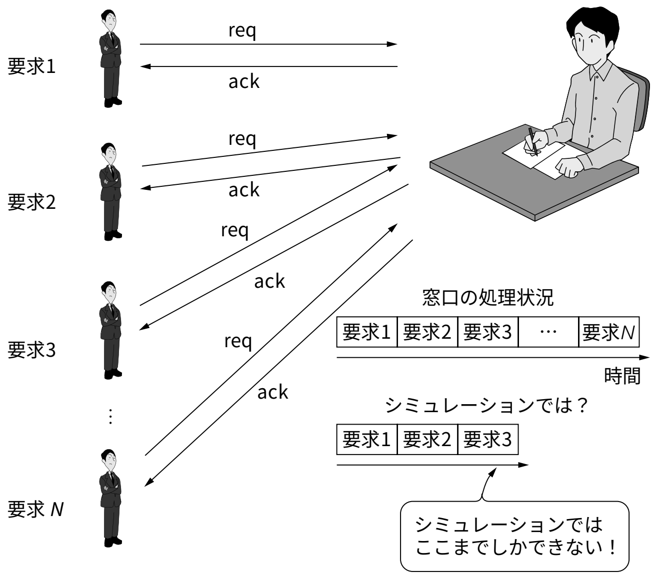
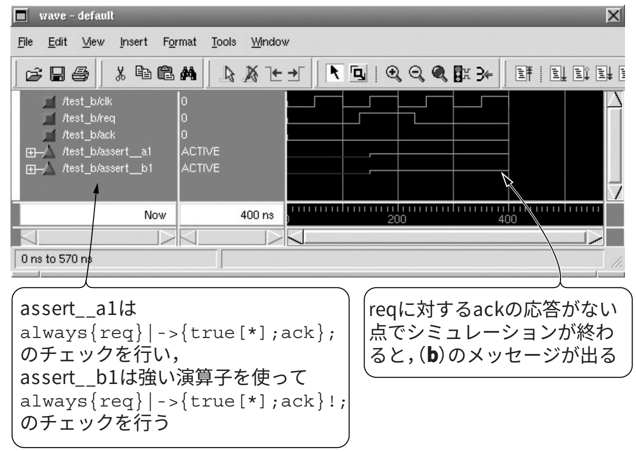


図8　シミュレーションなどで発生する状況

N箇所から要求が出た場合，現実的にはいつか処理が終わる．しかし，シミュレーションでは，すべての処理が終わるまでシミュレーションを実行できるかどうか定かではない．このような場合，強い演算子と弱い演算子を使い分けることでエラー判定を制御できる．



(a) reqに対するackの応答前にシミュレーションを終了

```
**Error:Assertion assert_b1(File:ファイル名 Line:行数)failed
  Time:400ns Iteration:2 Region:/test_b File:ファイル名
```

(b) 出力されるメッセージ

図9　処理が完了する前にシミュレーションを停止した場合

強い演算子でプロパティを記述する（! を利用する）と，アサーションの処理の途中でシミュレーションが終了した場合，エラーになる．弱い演算子による記述の場合には，何も出力されない．

行(ack)してもらえます．しかし，みなさんも月末になると銀行が混んで，行列に並ぶことが多いと思いますが，行列に並ぶと，いつ処理が実行されるのかわかりません．ただし，待っていると，いつかは処理してもらえるはずです．

このプロパティを記述すると，以下のようになります．

```
{req} |-> {true[*] ; ack} !
```

しかし，シミュレーションでは限られた時間で検証する必要があり，すべての処理をチェックすることが不可能な場合が少なくありません．本来であればすべての処理をチェックすることが理想ですが，現実には途中までしかシミュレーションできません．この場合，強い演算子で記述すると，シミュレーション終了時にアサーションがFailとなります．

これに対して，弱い演算子の記述は以下のようになります．

```
{req} |-> {true[*] ; ack}
```

動作の違いとしては，シミュレーション終了時にチェックしているアサーションがFailになりません．アサーションについてはPassまたはFailで判断するので，このような場合にどうするかなどについては，別途，決定する必要があります（例えば，FPGAで実現したプロトタイプを用いて，より長いパターンによる検証を実施するなど）．

この二つの差を，米国Mentor Graphics社の「ModelSim 6.0」で実行した場合の結果を用いて示します．**図9**はreqからackの応答が返る前にシミュレーションを終了した場合で，**図10**はreqからackの応答が返ったあとにシミュレーションを終了した場合です．

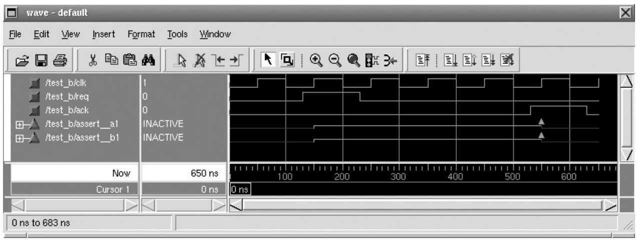
● プロパティ評価の論理式のキャンセルにはabortを利用

実際の設計では，ある処理が開始されたものの，途中でその処理がキャンセルされることがあります．この場合，アサーションのチェックを途中でやめるにはどうしたらよいのでしょうか？ これまでの説明だけでは対応できないと思います．

ここでプロパティ評価をキャンセルする方法を説明します．非同期リセットで2ビットのカウンタに対して，カウンタの値が1から2になることをアサーションでチェックします．リセットを考慮しないcountが1から2になるというプロパティは，以下のようになります．

```
property P1 is always {count = "01"}
                  |=> {count = "10"};
```

これに対して，カウンタの値が1から2になることをチェ



（**a**）reqに対するackを応答してから終了

```
**Note:Assertion assert__a1(File:req_ack_typeB.vhd Line:13)passed
  Time:550ns Iteration:1 Instance:/test_b
**Note:Assertion assert__b1(File:req_ack_typeB.vhd Line:15)passed
  Time:550ns Iteration:1 Instance:/test_b
```

（**b**）Passをログ出力したメッセージ

図10　処理が完了してシミュレーションを停止した場合
処理が完了していれば，強い演算子で記述したassert__b1も弱い演算子で記述したassert__a1も同じ結果になる．

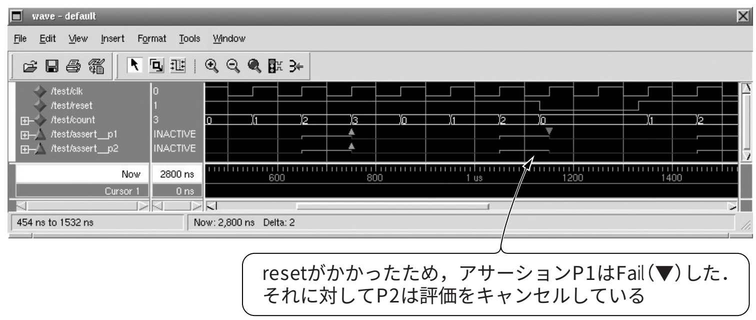


図11
途中で評価をキャンセルしたい場合
abortを使うと，リセットやエラーなどの例外時の動作によってアサーションの結果がFailになることが防げる．

ックはするが，リセットが0のときは評価しないという記述は，以下のように書きます．

```
property  P2  is  always  {count  =  "01"}
                          |=> {count = "11"}
abort reset = '0';
```

このように，abortでキャンセルする条件を記述します．この二つの記述でシミュレーションした結果を**図11**に示します．途中でリセットが有効になった時点で，P1をインスタンスしたアサーションではFailしています．一方，P2をインスタンスしたアサーションでは何も評価されていない（アサーションがキャンセルされた）状態になっています．abortを使うと，実際に用いられる処理に対して正確なアサーションを記述することができます．

● シーケンスのキャンセルにはendpointを利用

プロパティ評価をキャンセルするabortの条件に指定できるのは論理式だけで，シーケンスを持った条件は記述できません．例えば，resetが2サイクル有効になった場合にキャンセルするという記述を行うことができないため，使いかたに制限があります．

このような局面で使うことができる記述がendpointです．endpointでキャンセルするシーケンスを記述し，それをabortで利用することができます．endpointはキャンセルするかしないかを真または偽で返してくれるため，abortで使用可能となります．記述例を以下に示します．

```
endpoint end1 is {reset = '0' ;
                             reset = '0'};
property P2 is always {count = "01"}
                  |=> {count = "11"}
                               abort end1;
```

これまでに説明した内容で，かなり複雑なアサーションを記述できるはずです．今回説明した記述は，記事を読むだけでは理解しにくいかもしれませんが，実際にコードを書いてシミュレータで実行してみると理解しやすいものです．実際に手を動かしてみることが，習得への早道だと思います．

あかぼし・ひろき
(株)ロジック・リサーチ

<著者プロフィール>
赤星 博輝．ロジック・リサーチに所属しており，ハードウェアからソフトウェアまで担当するなんでも屋．もともとはEDAツールの開発を行っていたが，現在はハードウェア設計から徐々に検証技術に移っているところ．今後，どのようなしごとをするのかは，世の中の進みかたしだいです．